Model-independent differences

Könemann, Patrick

Published in:
ICSE Workshop on Comparison and Versioning of Software Models, 2009. CVSM '09

Link to article, DOI:
10.1109/CVSM.2009.5071720

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Model-Independent Differences*

Patrick Könemann
Technical University of Denmark, Informatics and Mathematical Modelling
Richard Petersens Plads, DK-2800 Kgs. Lyngby, Denmark
pk@imm.dtu.dk

Abstract

Computing differences (diffs) and merging different versions is well-known for text files, but for models it is a very young field – especially patches for models are still matter of research. Text-based and model-based diffs have different starting points because the semantics of their structure is fundamentally different. This paper reports on our ongoing work on model-independent diffs, i.e. a diff that does not directly refer to the models it was created from. Based on that, we present an idea of how the diff could be generalized, e.g. many atomic diffs are merged to a new, generalized diff. One use of these concepts could be a patch for models as it already exists for text files. The advantage of such a generalized diff compared to 'normal' diffs is that it is applicable to a higher variety of models.

1 Introduction

Text-based differencing and merging is used to compute differences between two different versions and merge them, e.g. if many developers are working on the same files. It is well-known in software development, but only applicable to text files. In the time of model-driven software engineering, models are used for software development and, in particular, for generating code. This makes differencing and merging for models desirable as well.

A textual diff can be stored as a *patch*, which is a self-containing file describing all differences between two versions of one or more text files. Its intention is to store the diff and make it applicable to other files, maybe on another workspace on which the original text files are not available. Moreover, a patch describes both states, before and after the change. This makes it possible during application of a patch to identify whether it was already applied or not; in addition, a patch can also be used in reverse direction, i.e. the changes can be undone.

*Supported by DTU grant; Project: Model-synchronization Technology for Model-Based Software Engineering

Support for differencing and merging of models is provided by some tools like RSA [6] and EMF Compare [1], and both are able to store the diff in a file for later reuse. However, it is always fixed to the models it was created from, hence it depends on the models and cannot be used similarly as patches for text files. This paper gives an overview of a way to *describe model-based diffs independently from the models they were created from*, hence such a model-independent diff can probably be used as a patch for models; further technical details are given in [5].

The paper is structured as follows. The overview in Sect. 2 motivates our work and gives the overall picture. Sect. 3 explains the key ideas of model-independent diffs, which are used in Sect. 4 for the generalization. Sect. 5 discusses related work and Sect. 6 concludes the paper.

2 Overview

This section outlines as challenges the main differences between text- and model-based diffs (structure, referencing, storing diffs) and derives the main requirements for model-based diffs (in italics).

Structure. A text-file always contains lines, each consisting of a string. However, models may consist of arbitrary elements, having attributes, references, and maybe other properties – so the structure of models may vary.
In order to support comparisons for models, we have to agree on a common meta model which describes the structure of our models. Our choice is the Eclipse Modeling Framework[1]*, because its meta model (ECore) is an implementation of EMOF, a subset of the* Meta Object Facility *(MOF, [8]) which is the basis for many modeling languages such as UML [10].*

Element referencing. In text files, each place can be addressed using a line number – but this is not the case for models. In some settings, each element might be addressed via a unique ID, but that depends on whether the meta model enforces unique IDs for each element. In any case, the struc-

[1] http://www.eclipse.org/modeling/emf

978-1-4244-3714-6/09/$25.00 © 2009 IEEE

ture of the model or the values of the elements can be used to match common elements.
If IDs are available, they would be the easiest way to address elements. Otherwise we need heuristics to find elements. The combination of the elements' structure and their attribute values is a very common strategy that is often used. But maybe this is not the best strategy either – we cannot decide a proper strategy for each model! To stay general, we propose an interface and some implementation ideas for referencing model elements as symbolic references; implementations may use e.g. IDs, heuristics, or other strategies.

Storing the addition of new model elements. A *patch* for text-files contains the added lines (i.e. strings) and some line numbers. This works, because lines are independent of the rest of the file. But that is not so easy for models for two reasons: first, what does it mean that an element is *contained* in a model. Second, model elements may have a more complex structure than just strings; they may have different attributes, maybe even sub-elements – hence we need to store sub-models. Third, the newly added element may contain references to other model elements, which is again the previous challenge. This problem does not occur in text files, because they do not have cross-references.
We need some kind of descriptor which sufficiently describes a sub-model, including multiple elements and references to other elements that may not be contained in this particular sub-model. It should work to use the same kind of symbolic references described previously.

Definition of terms

A *difference* (*diff* in short) describes structural changes made to a model which one can compute by comparing the model versions before and after the changes. Although it may consist of many small changes, we use the term *diff* to refer to all changes. A diff is *model-dependent* if it references the models (its versions resp.) it was created from. A *model-independent* diff, in contrast, is self-contained, i.e. the changes are described without refering to other models.

How model-independent diffs work

Next, we give the overall picture of our idea. We plan to use an existing differencing framework which already supports many differencing capabilities, e.g. EMF Compare [1], in order to create a model-dependent diff first. Then we focus on the model-independent representation of diffs, so we do not need to consider the diff creation process. Figure 1 gives an overview of the creation and application of model-independent diffs:

First, create a model-dependent diff from two versions of model *A* with an existing tool. The diff does not contain the actual differences, it just refers to the changed elements of

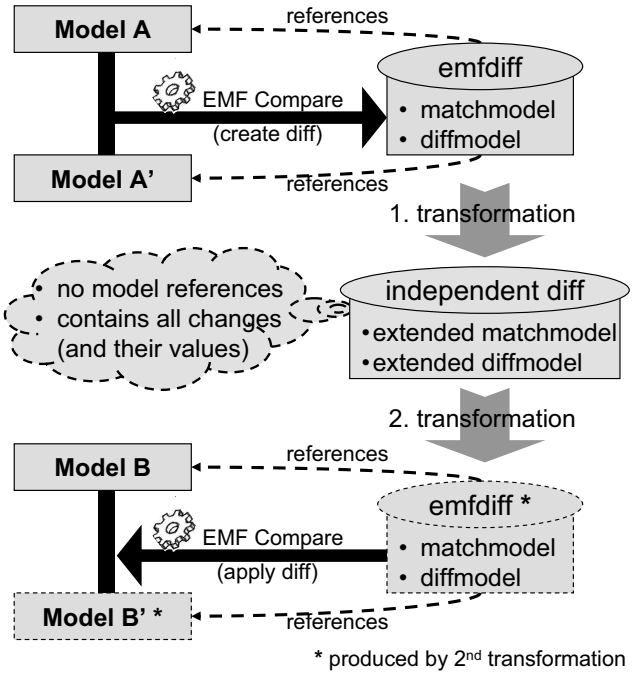

Figure 1. Transformations between model-dependent and model-independent diffs

the source and the target model, and gives some information about the difference itself.

Second, the 1st transformation is used to transform the model-dependent diff into a model-independent diff with respect to the three main challenges described above.

Third, in order to apply the diff to another model *B*, the references need to be restored and conflicts need to be identified. To do so, the 2nd transformation creates a new diff and a temporary model *B'*. Then, the same tool from the first step can be used to visualize and resolve potential conflicts.

The concepts covered in this paper focus on the properties of the first transformation and gives some thoughts how to realize the second transformation.

3 Differencing models

Due to the three challenges for model-independent diffs, we derived the following requirements for our work:

1. The model-independent diff must describe all differences independently from the originating models, i.e. it must contain the actual values which changed (EMF Compare, in contrast, refers to the models and describes *where* something was changed – the actual differences are implicit and are computed from the models on the fly).

2. It must contain information to find the changed element in arbitrary models; for instance by using IDs or struc-

tural properties.

3. We consider 9 types of changes: elements, attributes, and references, each may be changed, added, or removed.

Conflicts are not considered because we only want to store non-conflicting diffs. Later, if changes are applied to a particular model, there might be conflicts – but this is not of interest at the moment.

Example with unique IDs

Figure 2 shows a simple library meta model[2], in which books have a title and a catalogue number (which identifies a book uniquely). A model (in the middle of the figure) just contains one book titled *Galaxy*. On the right-hand side of the figure, the name of the book is changed to *Guide*.

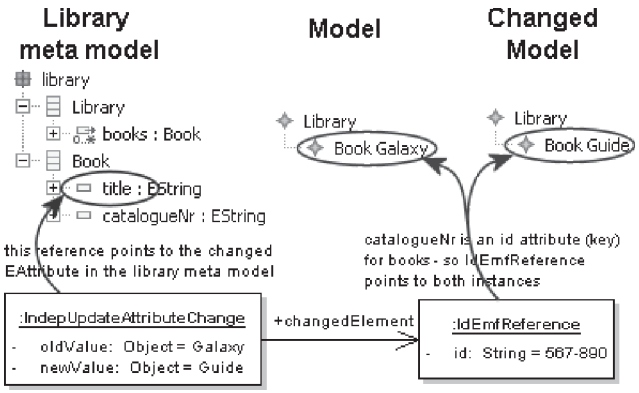

Figure 2. Parts of a model-independent diff for a changed attribute

The important part of the model-independent diff for this change is shown as an object diagram, containing an object of type *IndepAttributeChange*, which represents the actual change, and an object of type *IdEmfReference*, which we call a symbolic reference to the changed element. To be model-independent, the diff must not contain a direct reference to the models (as a patch for text files is also independent from the text files it was created from). So we need to store some information to point to the changed element, in this case the unique ID *catalogueNr* of the book.

The other parts of the diff are simple: the old as well as the new value of the changed attribute are stored in *oldValue* and *newValue*. The information of which attribute of the referenced element was changed, is given in a reference to the library meta model.

Our complete meta model for model-independent diffs is given in [5]. Next, we focus on how to use symbolic references without unique identifiers.

[2]In fact, this simple example shows a library *model*; however, our intention is to work with meta models later, e.g. UML.

3.1 Symbolic references

A model-independent diff needs to point to changed elements without directly referring to them. We use *symbolic references* (in literature also *indirect references*) to separate the diffs from the models. *"A symbolic reference is a character string that gives the name and possibly other information about the referenced item – enough information to uniquely identify [it]"* (from the book "Inside the Java Virtual Machine").

Unlike direct references, symbolic references do not require the referenced items to be available; however, a symbolic reference can be resolved to a direct reference which can be seen as a direct pointer to the de-referenced item. The example in Fig. 2 already motivated the need of such references. Instead of using a *character string*, we use the following meta model for describing symbolic references in model-independent diffs.

Meta model for symbolic references

As explained before in Sect. 2, there might be different ways of pointing to model elements. If elements have a unique ID, symbolic referencing can easily be done using the unique ID, which – by definition – identifies the element uniquely during its entire life-cycle. In the other case, we need *some other information about the referenced item*, for example its attribute values, some structural information, or its neighbour elements. The diagram in Fig. 3 indicates three possible implementations for symbolic references.

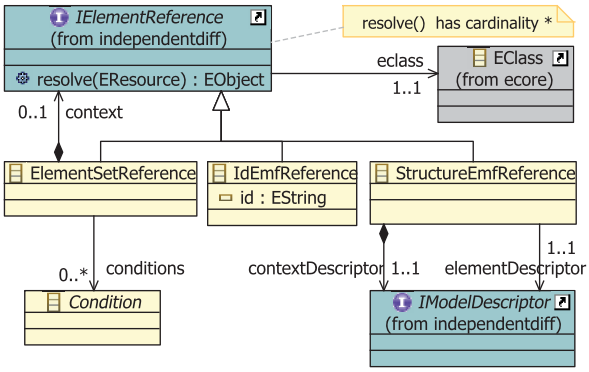

Figure 3. Meta model for symbolic references

The class *IdEmfReference* can be used to refer to elements which have an attribute marked as a unique identifier. For all other classes, we need to store *some other information*.The *ElementSetReference* has a set of conditions, e.g. in OCL (Object Constraint Language [9]), which can be used to identify one or a set of elements. This seems

to be the most flexible and powerful way referencing elements, due to the expressiveness of the conditions. We will see an example for that later in Sect. 4. The *StructureEmfReference*, on the other hand, contains a sub-model, which is supposed to contain sufficiant information to identify that particular item in a model. Then, the reference *elementDescriptor* points to the referenced item in the sub-model (an example for that is given in [5]).

3.2 Descriptor for sub-models

Model-independent diffs need to describe added elements, which may again contain other elements. The example in Fig. 4 shows such a case, two books with the stereotype *add* as well as some references have been added. There are actually two logical changes made to the model: first, a new book with a CD was added; second, a reference from the book *Galaxy* to the new book was added. Next we will see how these changes are expressed in a model-independent diff.

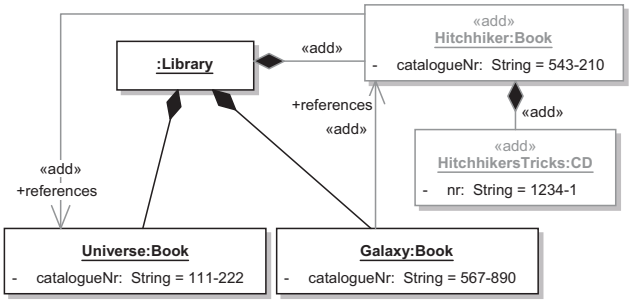

Figure 4. Two elements are added as a sub-model

We say that elements are *added* to a model, if they are *contained* in the later version but not in the earlier one. We consider containments as the main structure for all models, as it is the common case for EMF models. This has two important consequences: First, all changes concerning elements are based on containments, hence the addition, deletion, and movement of model elements. Second, although containments are a special type of references, they are not covered as reference changes.

With regard to the challenge in Sect. 2, there are several aspects we have to consider: 1. we have to store an entire hierarchy of model elements, 2. including their attribute values, and 3. also their references to other model elements.

According to our approach, both elements *Hitchhiker* and *HitchhikerTricks* are part of an addition and thus need to be described in a self-contained way in the diff, i.e. without references to the model in Fig. 4. Furthermore, it has to store the reference from *Hitchhiker* to *Universe*, because it is part of one of the newly added elements. Fig. 5 shows our meta model for such model descriptors.

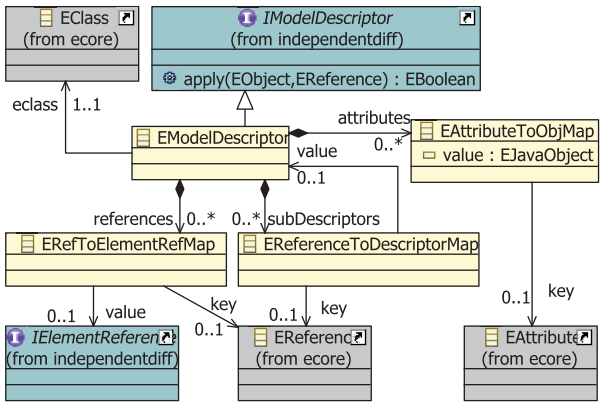

Figure 5. Meta model for sub-models

The reference from *EModelDescriptor* to *EClass* points to the type the particular model descriptor represents. In this case it would point to the EClass *Book* of the library meta model (cf. Fig. 2). Moreover, there are three maps for a model descriptor. The *ERefToElementRefMap* takes EReferences as a key (the reference from *Hitchhiker* to *Universe*, for example), and an IElementReference as the value. *IElementReferences* are symbolic references, which references model elements without directly pointing to them (cf. Sect. 3.1). The *EReferenceToDescriptorMap* takes containment references as keys and contains other model descriptors. In the example, it contains a descriptor for the element *HitchhikersTricks*. The *EAttributeToObjectMap* contains the values for all attributes of the particular element, in our example it needs to be the catalogueNr and the title. [5] explains that in more detail.

3.3 Change dependencies lead to groups

Unlike textual diffs, changes in models may *depend on each other*. The example in Sect. 3.2 contains two logical changes, a newly added sub-model and a reference from the book *Galaxy* to on of the new books. The second change obviously depends on the first one, because it uses one of its elements – consequently, the second change is useless if the first one is not applied before. We decided to use these dependencies to logically group changes in our diffs.

This is, of course, only a very simple example. But if we consider larger models with a lot of changes that depend on each other, then it would be nice to see structured groups which logically represent sets of independent changes. The idea is that these groups do not interfere with each other.

4 Generalization

Normal diffs contain all differences between two models, usually as a whole bunch of atomic changes. Fig. 6 again shows the library example with some books. There are three changes made which are structurally similar and each of them produces an entry in a diff. Three new references point from a customer to books, marked with the stereotype *add*.

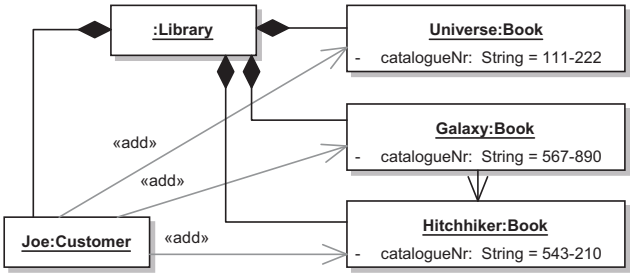

Figure 6. Similar changes: 3 new references

However, if this diff should be applied to another model (i.e. similar to a patch), it requires exactly these three books. So it is not possible to apply this diff to another model with different books, e.g. with different attributes. It might be useful in some cases to precisely refer to particular model elements, but in some cases it might be useful to weaken the precision of the target model elements in order to create a diff that can be used similarly as a patch. We could re-phrase our three changes to: *new references were added from the customer 'Joe' to all books in the library the customer belongs to.* Thus we weaken the description of the changes in such a way that it fits for all atomic changes at once, in order to find a more concise and adequat represenation of these changes. Consequently, we can probably apply this change to other models with other libraries even with different books. We call this kind of weakening *generalization of changes*.

Referencing sets of elements

Starting from the example above, we would like to have *one* change describing *many* references from a customer to *many* books. To do so, we allow symbolic references to not only refer to one particular but to a *set of elements*. Then we can combine these three changes (which only differ in the target of the reference) to one that describes a set of target elements. Fig. 7 shows shows that idea for the example:

IndepAddRefChange is such a combination and uses *ElementSetReference*s (cf. Sect. 3.1) as an implementation of symbolic references for this purpose. The three classes at the top are part of the library meta model; all other elements are concrete objects of a generalized diff as a UML object diagram. The curved arrows are references from the diff to the meta model as described by our diff meta model. The left-most *ElementSetReference* contains an OCL condition, which, if resolved via the context, returns the customer Joe. The context is yet another *ElementSetReference*, which resolves a library – here only one library exists. The right-most *ElementSetReference* resolves all books of that particular library, as described by its OCL condition *"self.oclIsKindOf(Book) and context.books.contains(self)"*, where *books* is the containment between the library and books at the top.

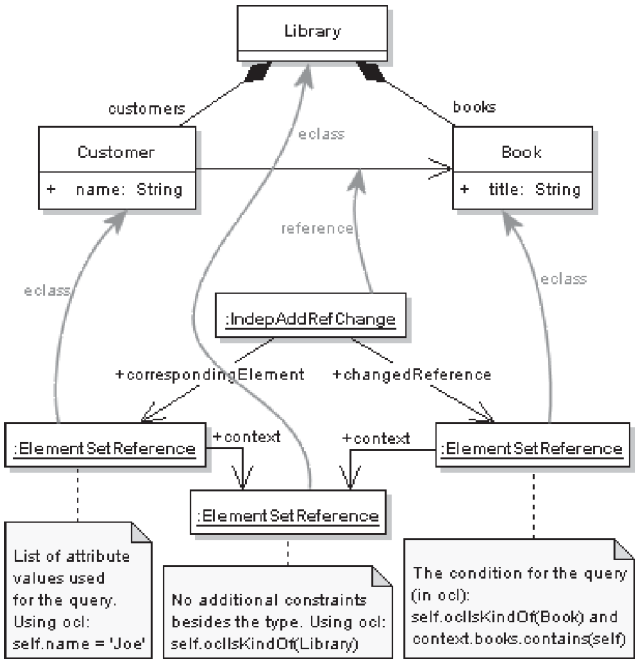

Figure 7. Symbolic references for resolving sets of elements

To summarize, resolving symbolic references to a set of elements allows us to generalize diffs. The key for this concept is the use of a language for describing such sets. In our example, we have used OCL conditions to describe the constraints the particular elements have to fulfill. However, there are important consequences compared to normal diffs:

Advantages: Model-independent diffs are possible with symbolic references and sub-model descriptors, because the diff does not any more depend on the models it was created from. By generalizing such a diff, it becomes more intuitive, concise, and compact, and it can even be the basis for a patching mechanism to a high variety of models – the reason is that the language for symbolic reference sets is powerful enough to resolve elements for different contexts.

Drawbacks: There is some overhead, probably even user interaction, required to construct such a diff. Moreover, we cannot say whether / how many of the generalized diffs have already been applied to a particular model, so we loose bidirectionality. Furthermore, if a change has more than one element set, the meaning of combining these sets is ambiguous.

5 Related Work

The concepts described here concern the representation of diffs in models without considering the graphical notation, e.g. diagrams. Most approaches deal with diff computation, merging, or concentrate on diagrams, but do not take model-independence into account [7, 3]. Though some tools already provide similar functionality.

In [4], we already presented and implemented a model-independent diff for a particular tool, namely the Enterprise Architect. It supports automatic diff creation as a list of changes, and a semi-automatic transfer (with conflict resolution) to other model versions. However, the diff was based on unique IDs and implemented only for the meta model of this particular tool. A transfer to another type of model is not possible because the concepts are hardcoded.

The main purpose of EMF Compare [1] is the support of differencing and merging for arbitrary models based on EMF. It provides a GUI for comparing and merging models, including conflict resolution. However, support for patches (which can be seen as model-independent diffs) is not yet included but scheduled for the next release.

The Rational Software Architect is also capable of differencing and merging UML models which includes difference detection, visualization, conflict resolution, and merging [6]. It relies on unique IDs, but also supports merging models without IDs (fusion of models). Similar to EMF Compare, it can store those diffs for later re-use, but they still refer to the originating models.

[2] has a similar goal but a different strategy. First, they extend each class in the meta model of the compared models with three new classes for the addition, deletion, and change of model elements. Second, the diff between two models is computed and stored according to the extended meta model. Third, they create higher-order model transformations on these extended meta models to transfer diffs to other models. So their approach works for arbitrary models. However, they did not consider conflicts so far, and their difference representation is not generalized.

6 Conclusion

The main contribution of this paper is the idea making diffs model-independent and generalizing them for making them applicable to a higher variety of models. Technical details and more example are given in [5].

Generalized changes provide two very interesting improvements over atomic changes: namely a more compact and concise form of describing sets of elements, as well as much broader application scenarios due to a powerful language describing element sets. On the other hand, there are some important drawbacks which may not be desirable in some cases – e.g. the loss of bidirectionality. To conclude, one needs to decide when to use which constructs for describing changes, depending on the context.

The next step is the application of generalized changes to other models as mentioned in Sect. 4. It is part of the second transformation outlined in Fig. 1 and subject of future work.

Acknowledgements

I would like to thank my supervisor Ekkart Kindler for many helpful discussions and advices.

References

[1] EMF Compare project. http://www.eclipse.org/emft/projects/compare, 2009.

[2] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[3] S. Förtsch and B. Westfechtel. Differencing and Merging of Software Diagrams – State of the Art and Challenges. In J. Filipe, M. Helfert, and B. Shishkov, editors, *International Conference on Software and Data Technologies (ICSOFT), Setubal (Portugal)*, volume 2. Institute for Systems and Technologies for Information, Control and Communication, 2007.

[4] E. Kindler, P. Könemann, and L. Unland. Diff-based model synchronization in an industrial mdd process. Technical Report IMM-Technical Report-2008-07, Technical University of Denmark, June 2008.

[5] P. Könemann. Model-independent diffs. Technical Report IMM-Technical Report-2008-20, Technical University of Denmark, Dec. 2008.

[6] K. Letkeman. Comparing and merging UML models in IBM Rational Software Architect. http://www.ibm.com/developerworks/rational/library/05/712_comp/, July 2005. 7 parts.

[7] T. Mens. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28(5):449–462, May 2002.

[8] Object Management Group. *Meta Object Facility (MOF) Core, V2.0*. http://www.omg.org/cgi-bin/doc?formal/2006-01-01, Jan. 2006.

[9] Object Management Group. *Object Constraint Language Specification, V2.0*. http://www.omg.org/cgi-bin/doc?formal/2006-05-01, May 2006.

[10] Object Management Group. *Unified Modeling Language, Superstructure, V2.1.2*. http://www.omg.org/cgi-bin/doc?formal/2007-11-02, Nov. 2007.